Verification of Security Protocols

Chapter 5: ProVerif — A Resolution Prover for Protocol Verification

Christian Haack

March 10, 2008

Plan for Today

- We will learn about a tool called ProVerif.
- ProVerif's input language is a generic variant of the spi-calculus.
- Unlike Cryptyc, ProVerif does not require type annotations.
- ProVerif fully automatically tries to prove that the protocol is robustly safe.
- There are three possible outcomes:
  1. ProVerif succeeds to prove robust safety.
  2. ProVerif finds an attack as a counterexample to robust safety.
  3. ProVerif can neither prove nor disprove robust safety.
- In rare cases, ProVerif does not terminate.

A Generic Spi-calculus

- ProVerif's input language is a generic variant of the spi-calculus.
- This input language allows users to define their own cryptographic primitives.

Generic Spi: Messages

Constructors.

f ranges over constructor symbols, each one with a fixed arity.

Messages.

M, N, L, K	$::=$	x	variable
	$\mid$	n	name
	$\mid$	$f(M_1, \ldots, M_n)$	constructor application

- Binary constructors for symmetric and asymmetric encryption:

  $encrypt(M, K)$: M encrypted with symmetric key K

  $pencrypt(M, K)$: M encrypted with encryption key K
- Constructors for n-tupling, hashing and key extraction:

  $ntuple(M_1, \ldots, M_n)$, $hash(M)$, $enc(K)$, $dec(K)$
- You can add more.

Generic Spi: Destructors

Destructors.

- g ranges over destructor symbols, each one with a fixed arity.
- For each destructor there is an associated set of equations of the following form:
  $g(M_1, \ldots, M_n) = M$
  where $M_1, \ldots, M_n, M$ do not contain names

- Symmetric key decryption: $decrypt(M, K)$
  - $decrypt(encrypt(x, y), y) = x$
- Asymmetric key decryption: $pdecrypt(M, K)$
  - $pdecrypt(pencrypt(x, enc(y)), dec(y)) = x$
- Projection: $ithOfn(M)$
  - $ithOfn(ntuple(x_1, \ldots, x_n)) = x_i$

Generic Spi: Processes

Processes.

$$
\begin{array}{lcl}
P, Q, R, O & ::= & \text{out } L\ M; P \\
& \mid & \text{inp } L\ M; P \\
& \mid & \text{stop} \\
& \mid & P \mid Q \\
& \mid & !P \\
& \mid & \text{new } n; P \\
& \mid & \text{let } x = g(M_1, \ldots, M_n) \text{ in } P \text{ else } Q
\end{array}
$$

- Conditionals, let-definitions and pattern-matching for tuples can be defined as derived forms.

Generic Spi: Operational Semantics

Destructors define partial functions.

- We write $\mathrm{def}(g)$ for the set of all equations for g.
- We say that $g(N_1, \ldots, N_n)$ is defined iff there exists an equation $g(M_1, \ldots, M_n) = M$ in $\mathrm{def}(g)$ and a substitution σ such that $\sigma M_i = N_i$ for all $i \in \{1 \ldots n\}$. In this case, we define $g(N_1, \ldots, N_n) = \sigma M$. Do example on board.

Operational semantics for destructors.

- let $x = g(M_1, \ldots, M_n)$ in P else $Q \rightarrow \{M/x\}P$
  if $g(M_1, \ldots, M_n) = M$
- let $x = g(M_1, \ldots, M_n)$ in P else $Q \rightarrow Q$
  if $g(M_1, \ldots, M_n)$ is undefined

- All other rules of the operational semantics are exactly as in the non-generic spi-calculus.

Generic Spi: An Example

A generates public message m

$A \rightarrow B \quad : \quad (m, \{hash(m)\}_k)$

In generic spi:

$$
\begin{array}{lcl}
P_A & \triangleq & \text{new } m; \text{out } net\ (m, encrypt(hash(m), k)); \\
P_B & \triangleq & \text{inp } net\ x; \\
 & & \text{let } (y_1, y_2) = x \text{ in let } z = decrypt(y_2, k) \text{ in} \\
 & & \text{if } z = hash(y_1) \text{ then stop}
\end{array}
$$

Executing $P_A \mid P_B$:

$$
\begin{array}{l}
P_A \mid P_B \\
\quad \rightarrow^* \quad \text{new } m; \text{let } z = decrypt(encrypt(hash(m), k), k) \text{ in} \\
\qquad\quad \text{if } z = hash(m) \text{ then stop} \\
//decrypt(encrypt(hash(m), k), k) = hash(m) \\
\quad \rightarrow \quad \text{new } m; \text{if } hash(m) = hash(m) \text{ then stop}
\end{array}
$$

ProVerif: Top-level Syntax

A ProVerif source consists of:

- a sequence of declarations
- a process

ProVerif sources.

$$\langle \textit{decl} \rangle^* \ \texttt{process} \ \langle \textit{process} \rangle$$

- Declarations declare free names, constructors, destructors, process macros, queries, and possibly other things.
- Processes are written in a version of the generic spi-calculus.

ProVerif: Names and Variables

Names.

- New-bound identifiers are names.
- Globally declared free identifiers are names.
- Undeclared free identifiers are names. (Avoid these!)

Variables.

- Let-bound identifiers are variables.
- Input-bound identifiers are variables.

Declare all your free names.

- ProVerif warns you when your model contains undeclared free names.
- Unfortunately, it is easy to miss these warnings, as they are at the very top of some very verbose output.
- Check for these warnings and get rid of them, as undeclared free names are almost always typos.

ProVerif: Declaring Free Names

Free names

- are public by default
- can optionally be declared private
- private free names are equivalent to names that are new-bound in front of the main process

Free name declarations.

$$[\texttt{private}]\ \texttt{free}\ \mathsf{id}_1, \ldots, \mathsf{id}_n.$$

Typical examples.

- untrusted channels (e.g. net)
- agent ids (e.g. server, spy)
- message tags

ProVerif: Declaring Constructors

Constructor declarations.

[`private`] `fun` id/*n*.

- This declares a constructor id of arity *n*.

Examples.

- `fun encrypt/2.`
- `fun sign/2.`
- `fun hash/1.`

Private constructors can be useful, too:

- `private fun serverkey/1.`
- `serverkey(Alice)` is the key that Alice shares with the server.
- The server can use this constructor to look up client keys.
- The attacker cannot use this constructor.

ProVerif: Declaring Destructors

Destructor declarations.

$$[\texttt{private}]\ \texttt{reduc}\ \mathsf{id}(\vec{M}) = N.$$

- This declares a destructor id with its reduction rule.
- The messages $\vec{M}, N$ must not contain names (but typically contain variables).
- It is also allowed to associate several reduction rules (separated by semicolons) with a single destructor.

Example.

- `reduc decrypt(encrypt(x,y),y) = x.`

ProVerif: Process Macros

Process macros.

`let` id = ⟨*process*⟩.

- After this declaration you can refer to the ⟨*process*⟩ by id.
- ProVerif textually replaces the id by the ⟨*process*⟩.

ProVerif: Pattern-matching

- ProVerif supports pattern-matching both at input and in let-expressions, but only against (possibly nested) tuples.
- To match you need to precede the id with $=$.
- To bind omit the $=$.

Examples.

- `let (=tag,=B,x) = decrypt(ctext,k) in ...`
- This pattern is matched by a triples `(tag,B,M)`, binding `M` to `x`.
- `let hash(x) = decrypt(ctext,k) in ...` No!
- This is not allowed, because it does not match against a (possibly nested) tuple.

ProVerif: Events

Events

$$\langle \textit{process} \rangle \quad ::= \quad \cdots \quad | \quad \texttt{event}\ M\ [;\langle \textit{process} \rangle] \quad | \quad \cdots$$

- Events can be inserted into processes.
- They generalize correspondence assertions.
- They have no effect at runtime, just like correspondence assertions.

Examples.

- `event beginSend(A,B,m)`
- `event endSend(A,B,m)`

ProVerif: Queries

In the declaration section, you need to query for the properties that you want ProVerif to analyze:

Secrecy.

$$\texttt{query attacker :}\ M.$$

Queries if the attacker can obtain M.

Many-one correspondence.

$$\texttt{query ev :}\ M\ \texttt{==> ev :}\ N.$$

Queries if event M is always preceded by event N.

One-one correspondence.

$$\texttt{query evinj :}\ M\ \texttt{==> evinj :}\ N.$$

Queries if event M is always preceded by event N, and every trace contains at least as many N-events as M-events.

ProVerif: Names in Queries

If a query mentions a name (i.e., an identifier that is declared free or new-bound), this name stands for itself.

Example.

```
query attacker : s.

process
  ...
  new s;
  ...
```

- Note that, in a top-level query, you can mention a name that is generated deep inside a process. (Weird scoping.)
- To prevent ambiguities, ProVerif warns you when you use the same name in two different new-binders.

ProVerif: Variables in Queries

Variables in correspondence queries are implicitly universally quantified.

Example.

```
query ev:endSend(x,y,z) ==> ev:beginSend(x,y,z).
```

This is like:

```
query (∀ x,y,z)
  (ev:endSend(x,y,z) ==> ev:beginSend(x,y,z)).
```

This is probably what you want.

ProVerif: Variables in Queries

Variables in secrecy queries are implicitly existentially quantified.

Example.

```
query attacker : x.
```

This is like:

```
query (∃ x)(attacker : x).
```

This is always true and probably not what you want. You usually want to query for secrecy of names, not variables.

ProVerif: Modeling Internal Threats

This is done as always:

- include a compromised agent called spy
- publish the spy's secrets

Furthermore, we need to use conditional end-events:

```
...
( if a<>spy then event endSend(a,b,m) ) |
...
```

instead of

```
...
event endSend(a,b,m);
...
```

Why? The latter is trivially violated for `a=spy`.

ProVerif: Conditional Secrecy

Recall our conditional secrecy assertions:

```
...
if a<>spy then secret(M) |
...
```

How do we represent these in ProVerif? Like this:

```
query attacker : flag.
...
( new flag; if a<>spy then out(M, flag) ) |
...
```

ProVerif: Conditional Secrecy

```
query attacker : flag.
...
( new flag; if a<>spy then out(M, flag) ) |
...
```

Why does this encoding work?

- If the attacker learns M, then he also learns flag because flag gets published on channel M.
- If the attacker learns flag, then he can only have learned it through channel M (because flag does not occur anywhere else in the program). So the attacker must know M.
- The second argument only works if we know that M is never used as a receiving channel. This is for instance the case for protocols that use only global channels (like net), as it is often the case in our protocol models.

ProVerif: Attack Traces

- ProVerif's attack traces are sequences of:
  - output events
  - input events
  - correspondence events
- The attacker's input/output events are omitted.
- Often attack traces are contaminated by initialization messages.
- Names found in the program text are subscripted by numbers. This is because a single name in the program text corresponds to many names at runtime (if inside a replication).

ProVerif: Pitfalls to Avoid

- Often typos result in unreachable code.
- Unreachable code is safe.
- ProVerif does not check for typos.
- Your protocol may verify due to a typo!

To prevent this:

- Declare all free names. (Check for warnings at the very top of the output.)
- Test if you reach the end of the protocol by inserting a secrecy violation at the end.

```
query attacker : typoflag.
...
new typoflag; out(net, typoflag)
(* end of the protocol *)
```

If the end of the protocol is reachable, ProVerif finds the obvious attack on this.

From Spi-calculus to Logic Formulas

- Now, we will sketch how ProVerif translates spi-calculus processes to a set of logic formulas.
- Via this translation, ProVerif reduces the secrecy problem for cryptographic protocols to a logical problem.
- The translation is sound: If no secrecy violation can be logically derived from the formula set that results from translating protocol P, then P is robustly safe for secrecy.
- The translation is incomplete: If a secrecy violation can be logically derived from the set of formulas that results from translating protocol P, then P may in rare cases still be robustly safe for secrecy.

Horn Clauses

- Processes are translated to sets of Horn clauses.
- Horn clauses are predicate logic formulas of the following form:

$$(\forall \vec{x})(P_1(\vec{M}_1) \wedge \cdots \wedge P_n(\vec{M}_n) \Rightarrow Q(\vec{N}))$$

  where $n \geq 0$ and $P_1, \ldots, P_n, Q$ are predicate symbols.
- The universal quantifier is usually left implicit.
- Resolution provers take a set of Horn clauses and a goal $(\exists \vec{x})(P(\vec{M}))$ and check if this goal is provable from the clauses.
- ProVerif is based on a resolution prover that is tailored for protocol verification.

Facts

The translation from spi targets a formula language with only two predicates:

Facts.

F	::=		facts
		$attacker(M)$	attacker knowledge
	|	$mess(L, M)$	channel messages

- The unary predicate *attacker* has the following meaning:

  $attacker(M)$: The attacker knows M

- The binary predicate *mess* with the following meaning:

  $mess(L, M)$: Message M is out on channel L

The Structure of the Translation

- Input:
  - a closed process P
  - a subset S of P's free names representing public names (all names except those declared `private`)
- Output:
  - a set $B(P, S)$ of Horn clauses
- The top-level definition:
  - $B(P, S) \triangleq \mathit{InitialAttackerKnowledge}(S) \cup \mathit{AttackerRules} \cup \mathit{ProtocolRules}(P)$

Initial Attacker Knowledge

The initial attacker knowledge is simple:

$$\mathit{InitialAttackerKnowledge}(S) \triangleq \{ \mathit{attacker}(n) \mid n \in S \}$$

The Attacker Rules

The attacker rules only depend on the choice of constructors and destructors.

Show attacker rules for sym. crypto on blackboard in parallel

- For each constructor f:

  $$attacker(x_1) \wedge \cdots attacker(x_n) \Rightarrow attacker(f(x_1, \ldots, x_n))$$

- For each destructor g,
  for each equation $g(M_1, \ldots, M_n) = M$ in def(g):

  $$attacker(M_1) \wedge \cdots attacker(M_n) \Rightarrow attacker(M)$$

- $mess(x, y) \wedge attacker(x) \Rightarrow attacker(y)$
  - In words: If message y is out on channel x and the attacker knows channel x, then the attacker knows message y.
- $attacker(x) \wedge attacker(y) \Rightarrow mess(x, y)$
  - In words: If the attacker knows channel x and message y, then the attacker can output message y on channel x.

The Protocol Rules: The Basic Idea

Each output statement out c N generates a Horn clause of the following form:

- $mess(c_1, M_1) \wedge \cdots \wedge mess(c_n, M_n) \Rightarrow mess(c, N)$
  where $M_1, \ldots, M_n$ are the messages that have been received previously

Examples.

- $P \triangleq$ inp net x; inp net y; out net (x, y)
  - $ProtocolRules(P) =$
    $\{ mess(net, x) \wedge mess(net, y) \Rightarrow mess(net, (x, y)) \}$
- $Q \triangleq$ inp net x; let $y = decrypt(x, k)$ in out net y
  - $ProtocolRules(Q) =$
    $\{ mess(net, encrypt(y, k)) \Rightarrow mess(net, y) \}$

See [AB05] for the details of the translation.

Secrecy

Definition (Secrecy [AB05]).

Let S be a finite set of names.

- The closed process O is an S-adversary iff its free names are contained in S.
- The closed process P preserves the secrecy of n from S iff $P \mid O$ does not output M on c for any c in S.

- This secrecy definition is slightly different from our earlier definition from class.
- The differences are:
  1. It defines secrecy for names only (not for variables).
  2. It defines secrecy for free names, but not for new-bound names.
- The above definition could be adapted to define secrecy for new-bound names, but it would make the statement of the soundness theorem slightly uglier.

Soundness of the Logic Translation

Soundness Theorem

Let P be a closed process and $s \notin S$. If $attacker(s)$ cannot be logically derived from $B(P, S)$, then P preserves the secrecy of s from S.

- Abadi and Blanchet's proof of this theorem is interesting.
- It translates $B(P, S)$ to an instance of a generic type system and relies on a soundness theorem for the type system. See [AB05] for details, if interested.

Horn Clauses for an Unsafe Protocol

$$A \rightarrow B \quad : \quad s$$

This process does not preserve secrecy of s from $\{net\}$.

Horn clauses $B(P, \{net\})$.

- Initial attacker knowledge:
  - $attacker(net)$
- Attacker rules:
  - $mess(x, y) \wedge attacker(x) \Rightarrow attacker(y)$
  - $attacker(x) \wedge attacker(y) \Rightarrow mess(x, y)$
- Protocol rules:
  - $mess(net, s)$

- $attacker(s)$ is logically derivable from these rules.
- And here is how:
  - From $mess(net, s)$ and $attacker(net)$ and the first attacker rule, it follows that $attacker(s)$, by logical implication.

Horn Clauses for a Safe Protocol

$$A \to B \quad : \quad \{s\}_k$$

This process preserves secrecy of s from $\{net\}$.

Horn clauses $B(P, \{net\})$.

- Initial attacker knowledge:
  - $attacker(net)$
- Attacker rules:
  - $mess(x, y) \wedge attacker(x) \Rightarrow attacker(y)$
  - $attacker(x) \wedge attacker(y) \Rightarrow mess(x, y)$
  - $attacker(x) \wedge attacker(y) \Rightarrow attacker(encrypt(x, y))$
  - $attacker(encrypt(x, y)) \wedge attacker(y) \Rightarrow attacker(x)$
- Protocol rules:
  - $mess(net, encrypt(s, k))$

- $attacker(s)$ is not logically derivable from these rules.
- Why not? We'll see next time how ProVerif checks this systematically.

Looking at ProVerif's Horn Clauses

- At the top of ProVerif's output, you always see a sequence of numbered rules. This is the Horn clause representation of your spi-calculus protocol.
- ProVerif seems to use a slightly different translation that makes more use of the attacker predicate and less use of the mess predicate.
- The translation I sketched is from Abadi and Blanchet's paper [AB05].

Why do Names have Arguments?

- When looking at ProVerif's output, you will notice that each name is associated with a (possibly empty) list of arguments, e.g., $n[x, y, c]$.
- The arguments record messages that existed before n got generated.
- This information helps the theorem prover to rule out impossibilities.

Example.

- $P \triangleq \text{inp } net\ x; \text{inp } net\ y; \text{new } n; P$
- When translating P, the name n gets represented by:

$$n[x, y]$$

- The arguments indicate that x and y existed before n got generated.
- The prover now knows that $x = n[x, y]$ is impossible because n has been generated after x.

Homework

I have posted a homework assignment using ProVerif. It's due on Monday, 31 March.

References

Martin Abadi and Bruno Blanchet.
Analyzing security protocols with secrecy types and logic programs.
Jounal of the ACM, 52(1):102–146, 2005.

Bruno Blanchet.
ProVerif User Manual.